FAULT PROPAGATION ANALYSIS OF LARGE-SCALE, NETWORKED EMBEDDED SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Aliva Pattnaik

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
College of Computing

Georgia Institute of Technology
December 2011

FAULT PROPAGATION ANALYSIS OF LARGE-SCALE, NETWORKED EMBEDDED SYSTEMS

Approved by:

Dr. Mary Jean Harrold, Advisor
College of Computing
Georgia Institute of Technology

Dr. Alessandro Orso
College of Computing
Georgia Institute of Technology

Dr. Mayur H. Naik
College of Computing
Georgia Institute of Technology

Dr. Shamkant B. Navathe
College of Computing
Georgia Institute of Technology

Date Approved: November 14, 2011

To all the graduate students who work very hard under many constraints.

ACKNOWLEDGEMENTS

I would like to thank my husband, Saswat Anand, for his constant guidance, encouragement, and enormous support through my Masters degree. I extend my sincere thanks to my advisor Dr. Mary Jean Harrold for standing beside me during my difficult times. I had an extremely pleasing experience while working with her, both personally and intellectually. I would like to thank Boeing Commercial Airplanes for funding this research. I would like to thank Mr. John Joseph Chilenski and Ms. Raquel S. Whittlesey-Harris for answering my numerous questions during this work. I would like to thank members of the Aristotle Research Group for allowing me to come to the lab with my son, and even babysitting him on several occasions. I would like to thank my committee—Dr. Mary Jean Harrold, Dr. Alessandro Orso, Dr. Mayur Naik, and Dr. Shamkant Navathe—for their time and care in reading and reviewing my thesis. Finally, I would like to thank my father Mr. Biswanath Pattnaik for his constant encouragement and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AFDX	Avionics Full Duplex Switched Ethernet
ARINC	Aeronautical Radio, Incorporated
CCS	Common Core System
CDN	Common Data Network
CSML	Communication System Markup Language
FauPA	Fault Propagation Analyzer
FMEA	Failure Mode and Effects Analysis
FTA	Fault Tree Analysis
IMA	Integrated Modular Avionics
LRU	Line Replaceable Unit
RDC	Remote Data Concentrator
TTP	Time-Triggered Protocol
VL	Virtual Link

SUMMARY

In safety-critical, networked embedded systems, it is important that the way in which a fault(s) in one component of the system can propagate throughout the system to other components is analyzed correctly. Many real-world systems, such as modern aircrafts and automobiles, use large-scale networked embedded systems with complex behavior. In this work, we have developed techniques and a software tool, FauPA, that uses those techniques to automate fault-propagation analysis of large-scale, networked embedded systems such as those used in modern aircraft. This work makes three main contributions.

1. Fault propagation analyses. We developed algorithms for two types of analyses: forward analysis and backward analysis. For backward analysis, we developed two techniques: a naive algorithm and an algorithm that uses Datalog.
2. A system description language. We developed a language that we call Communication System Markup Language (CSML) based on XML. A system can be specified concisely and at a high-level in CSML.
3. A GUI-based display of the system and analysis results. We developed a GUI to visualize the system that is specified in CSML. The GUI also lets the user visualize the results of fault-propagation analyses.

CHAPTER 1

INTRODUCTION

Embedded systems that contain simple and inexpensive computational units have been widely-used for decades. However, in recent years, embedded systems have become dramatically more powerful and complicated. Large-scale, networked embedded systems in many domains (e.g., avionics, automobile, and bio-medical) use computing units that have the power of desktop computers, and different components of those systems communicate with each other over complex data networks (e.g., AFDX [1,8], FlexRay [3], and TTP [6]). In these systems, environmental data are collected by a large number of sensors, and then transmitted to computing units that process them. After processing, the output signals are sent over the data networks to effectors that influence physical processes. Examples of such embedded systems include the Boeing 787 and Airbus 380 commercial airplanes and NASA's Crew Exploration Vehicle that use AFDX protocol [7], Lockheed Martin's F-16 fighter planes that use TTP protocol [6], and BMW's X5 Sports Activity Vehicles that use FlexRay protocol [3].

In such systems, it is important to verify that data collection, transmission, and processing are performed correctly because such systems are often used in safety-critical domains, such as aircraft flight control, medical devices, and nuclear systems. Failure of such systems could result in loss of life, significant property damage, or damage to the environment. A fault in a sensor can cause faulty data to enter the system. A fault in a network device (e.g., a switch or a wire) or an inadequate redundancy measure can cause complete loss of data or unexpected delay in data reaching its destination. A fault in a computational unit, at either the hardware or software level, can cause the computation of faulty data.

1.1 Fault-Propagation Analysis

To gain confidence in the correctness of such systems, fault-propagation analysis is typically performed. To perform the analysis, a fault-propagation model is designed so that it represents whether a fault in one component or a combination of components can cause a critical component of the system to fail. The analysis then uses the fault-propagation model to determine whether there are any conditions under which faults in one or more components can cause the system to fail.

Fault-propagation analysis of modern large-scale networked embedded systems is more challenging than it was for older systems because these newer systems use shared network and computational components. In the avionics domain, older aircraft were designed based on a federated system architecture, in which different operations of the aircraft are controlled by different computing units, and computing units communicate using point-to-point protocols that do not share the network (e.g., ARINC 429 [5]). The advantage of a federated system is that failure from one computing unit cannot propagate to another unit because they are physically separated. However, such systems are heavy and expensive because for each operation there are dedicated resources such as wiring and computing units. In contrast, modern aircrafts are designed based on Integrated Modular Architecture (IMA) [9]. In IMA, a single central computing system controls all operations of the aircraft, such as avionics, environmental control, electrical, mechanical, hydraulic, auxiliary power unit, cabin services, flight controls, health management, fuel, payloads, and propulsion. Different components of an IMA-based system communicate using network protocols that share the communication network (e.g., AFDX, FlexRay, or TTP). Because resources (e.g., computing and networking devices) are shared among different operations, IMA-based systems are lighter and less expensive. The disadvantage of IMA architecture is that failure in one component of the computing system can affect many different operations of the aircraft because the component may be

shared. Thus, a thorough failure analysis of an IMA-based system is necessary.

One approach to performing failure analysis is to use a standard analysis, such as fault-tree analysis (FTA). FTA analysis requires that domain engineers create a fault-tree representation of the embedded system. In a *fault-tree*, the root node represents the specific undesired state that is being analyzed, and each leaf node represents a failure mode in the system. Each internal node represents an undesired system state that may arise as a result of one or more failures. The children nodes of an internal node represent states that aggregately cause the state represented by the internal node. The aggregation function is represented by logical gates, such as AND, OR, and XOR.

Manually creating a fault-tree representation of a system requires intimate understanding of the system. A fault-propagation model (e.g., fault tree) of a system is not specific to any domain, and thus, it requires modeling domain-specific concepts of the system at a lower-level of abstraction (e.g., AND and OR gates). As a result, creating a fault-tree representation of a large system can be error-prone and can require a great deal of manual effort. This manual effort can be significantly reduced if the fault-tree representation of a system can be automatically generated from a high-level, domain-specific description of the system.

1.2 Our Approach

To address the limitations of existing techniques, in this research, we have developed techniques that reduce the manual effort required to perform fault-propagation analysis of large-scale, networked embedded systems. We have developed two types of analyses: forward analysis and backward analysis. Forward analysis is used to simulate "what if" scenarios. Given a set of components that could be faulty, forward analysis determines whether a fault can propagate to any of the critical components. Whereas forward analysis is targeted towards experts who want to analyze and gain insights into the system's behavior, backward analysis exhaustively and fully-automatically analyzes

the system, and computes sets of components such that if all components in any of those sets become faulty, the fault can propagate to one or more of the critical components.

We have also developed an XML-based language, which we call Communication System Markup Language (CSML). CSML can be used to specify networked embedded systems that use AFDX data networks. Unlike the fault-tree representation, in CSML, there is no notion of logical gates, such as AND and OR. Instead, CSML supports AFDX-network terminology, such as virtual links, channels, and constructs, to specify commonly-used concepts of a networked embedded system, such as sensors, effectors, messages, and datasets.

1.3 FauPA: A Software System for Fault-Propagation Analysis

We implemented our fault-propagation analysis, along with our new XML-based language, CSML, in a system that we call FauPA (Fault Propagation Analyzer). Using our new techniques, FauPA reduces the time-consuming manual effort in performing fault-propagation analysis by automatically creating the fault-propagation model (which is similar to a fault tree) used for the analysis from the system specification. Using the model, FauPA supports various forms of verification on the entire system, such as those supported by FTA and FMEA. FauPA has three major components:

- **Fault Propagation Analyses**

  FauPA implements our fault-propagation analysis to check whether faults from a specific set of components can propagate to designated critical components (forward analysis), and also identifies the set of components from which fault can propagate to a designated critical component (backward analysis).

- **Communication System Markup Language (CSML)**

  FauPA inputs the description of the system model in a language that we call Communication System Markup Language (CSML), an XML based language. User can specify the model in CSML using domain-specific constructs supported by

CSML. FauPA automatically generates a fault-propagation graph from the CSML specification of the model. The fault-propagation graph is used in the subsequent analysis stage.

- **GUI-based Display**

FauPA's GUI displays the systems and the results of the analysis to the user. Information, such as the components through which the fault has propagated to the critical components, is shown on a graphical representation of the communication network. FauPA's GUI has features, such as zooming and panning that make the visualization of the system more comprehensible.

1.4 Advantages of Our Approach Over the Fault-Tree Analysis Approach

There are three important advantages of FauPA over FTA tools. The first advantage of FauPA over FTA tools is that FauPA does not require users to specify the fault tree for failure analysis. Construction of a fault tree requires users to have deep knowledge of the system. In particular, specifying the logical gates requires knowledge of how failure modes or undesired system states can give rise to other states. Additionally, it is cumbersome to create the fault tree for a system that has a large number of failure modes and states. Because FauPA is specifically designed to analyze a computing system that uses an AFDX data network, dependencies between failure modes and system states are automatically inferred and a corresponding fault tree is constructed from a specification of the physical system. This specification is easier to write than a specification of a fault tree, which does not resemble the physical form of the system.

The second advantage of FauPA over FTA tools is that it is more efficient than those tools. In failure analysis of computing systems, FTA can be used to check whether a specific critical component of the system can fail as a result of failure of other components. When the number of critical components is large and the goal is to check whether any of the critical components can fail, FTA must be repeated for each critical

component. In contrast, FauPA can check whether any of the critical components fail in one execution, which is more efficient than multiple FTA runs.

The third advantage of FauPA is that it is more readily applicable to IMA-based systems than FTA tools. Sharing of components in an IMA-based system leads to a graph topology of the system. A component that is shared by and receives data from multiple components is represented by a node that has multiple incoming edges in the fault-propagation graph. Because FTA analysis is performed over a fault tree, to analyze an IMA-based system, users need to convert the fault-propagation graph into a tree by duplicating nodes of the graph. In contrast, FauPA automatically creates the fault-propagation graph from the system description.

1.5 Overview of the Thesis

The rest of the thesis is organized as follows. Chapter 2 presents background information about the architecture of computing systems used in modern aircraft. Chapter 3 describes the fault-propagation analyses by providing details of the forward and backward fault-propagation analyses. Two alternative implementations of backward analysis are described: one alternative uses Datalog [11,13] and the other does not. Chapter 4 describes the Communication System Markup Language (CSML), an XML based language for specifying an AFDX-based communication system. This chapter also contains the complete CSML specification of a small system. Chapter 5 describes the different features of FauPA's GUI-based display of the system and analysis results. Chapter 6 concludes by presenting a summary of the thesis, merits of the research, and future work.

CHAPTER 2

COMPUTING SYSTEMS OF MODERN AIRCRAFTS

Although our overall goal is to develop techniques that automate fault-propagation analyses for large-scale, networked embedded systems, for this research we concentrate on systems that are similar to the core computing system (CCS) used in Boeing's 787 series aircrafts. The target system consists of four types of elements.

1. *Sensors and effectors* lie at the boundary of the system. Sensors are devices that take readings of environmental parameters, such as temperature and pressure, and effectors are devices that typically control a device by sending an output signal.
2. *Remote Data Concentrators* (RDC) consolidates inputs from sensors and delivers output signals to effectors.
3. The computing units that host processing and power control modules and network switches lie at the heart of computing system. The processing units receive data from RDCs, process them and when necessary, send output signal to RDCs.
4. An *Avionics Full Duplex Switched Ethernet* (AFDX) [1] based data network data network that transfers data between the computing units and RDCs.

The RDCs reduce the amount of wiring. Instead of each function being linked (often by long wires) directly to the CCS, a group of functions is connected by short wires to the local RDC, which, in turn, is linked to the CCS using an AFDX cable. The use of RDCs is similar to the use of USB-based devices on a personal computer, but now they are distributed.

AFDX, which is also known as the ARINC 664 (Part 7) standard, is a specification developed by ARINC Corporation for data networks used in aeronautical, railway, and military systems. The AFDX [1] specification is based on standard IEEE

802.3 Ethernet technology. AFDX extends the Ethernet standard by adding quality of service and deterministic behavior with a guaranteed dedicated bandwidth. In addition to Boeing 787 Dreamliner series of aircrafts, AFDX is currently used in the Airbus A380 and A400M.

There are two main advantages of AFDX over standard network technologies: (1) reliable packet transport and (2) bounded transport latency. These are the base requirements for a network for avionics applications. Standard Ethernet technology uses a topology in which each node in the network is treated equally. If a peer wants to transmit a packet on the network and the media is occupied, a collision occurs, the peer backs up and tries again, until the transmission is successful or a predefined amount of time has elapsed. This behavior introduces variable-length latency, which is not tolerable in safety-critical applications. However, these collisions can be avoided in an AFDX network because of its switched full-duplex topology.

To increase the availability of the network, AFDX is built using redundancy on the physical layer. Each data packet is transmitted simultaneously by two Ethernet controllers onto separate wires, using physically separate switches, to the destination system. The two redundant physical paths together form a logical connection called *Virtual Link* (VL). Because of the use of the VL, the network provides two properties that are important for interconnecting avionic systems with different levels of criticality: (1) a unidirectional private line with bounded latency and (2) guaranteed bandwidth. A single VL may have point-to-point or point-to-multi-point connections. The advantages of AFDX arise because it is compatible at the application level and saves a large number of cable runs by multiplexing many individual VLs onto a single wire connection. Because the IEEE 802.3 standard allows multicast packet transmission on the Ethernet, it is possible to use the standard mechanism to allow a packet to be routed to multiple destination nodes.

CHAPTER 3

FAULT PROPAGATION ANALYSES

In this chapter, we describe the forward and backward fault-propagation analyses, along with two alternative implementations of backward analysis: one alternative uses Datalog and the other does not.

3.1 Fault-Propagation Model

In FauPA, a fault-propagation model is automatically extracted from a high-level description of the system. The *fault-propagation model* is a directed graph that represents the way in which faults propagate through the system. A *node* in the graph corresponds to either one physical entity (e.g., sensor, RDC, value of a physical parameter, wire, or bus) of the model or a group of physical entities (e.g., channel or virtual link). We refer to the second type of nodes as *auxiliary nodes*. Each node has a binary *fault status*, which can be TRUE or FALSE: TRUE fault status of a node means that the node is faulty; FALSE fault status means that the node is not faulty.

If the fault status for a node corresponding to a physical entity is TRUE, a fault has been introduced in that entity. The fault status of an auxiliary node in the graph represents the collective fault statuses of the corresponding group of physical entities. For example, the fault status of an auxiliary node corresponding to a channel, which consists of switches and wires, represents whether those switches and wires can correctly transmit some data. If the fault status of such a node corresponding to a channel is TRUE, at least one of the switches or wires that belong to the channel is faulty. If the fault status is FALSE, none of the switches and wires is faulty. Furthermore, if the fault status of an auxiliary node is TRUE, the fault has propagated to this node from a node that corresponds to a physical entity.

An *edge* from node X to an auxiliary node Y means that the fault may propagate from X to Y, depending on Y's transfer function, which determines how Y's fault status is affected by X's fault status. An auxiliary node can have one of two types of *transfer functions*: OR transformation function or AND transformation function. The fault status of an auxiliary node that has an OR transfer function becomes TRUE if the fault status of any of its predecessors is TRUE, whereas, the fault status of an auxiliary node that has OR transfer function becomes TRUE if the fault statuses of all of its predecessors are TRUE. The nodes with the AND transfer function arise because of the system's redundancy measures. For example, a virtual link that has two redundant channels will become faulty only if both of its channels become faulty. Thus, a virtual link is represented by an auxiliary node with the OR transfer function.

Based on the criticality of the components in the model, the nodes in the graph are categorized as critical and non-critical nodes. *Critical nodes* are those that should never become faulty because the result would be catastrophic. In contrast, *non-critical nodes* can become faulty because they would not cause a catastrophic failure. The goal of the fault-propagation analysis is to exhaustively analyze whether faults from non-critical nodes can propagate and eventually make any of the critical nodes faulty.

3.2 Example of Fault-Propagation Graph

Figure 3.1 shows an example system. In Figure 3.1, VL_1 is not a physical entity in the model but consists of two physical paths: $Path_A$ and $Path_B$. Each path consists of different switches and wires to connect them. $Path_A$ is (a_1, $wire_1$, a_2) and $Path_B$ is (b_1, $wire_2$, b). $Reading_1$ and $Signal\_out_1$ are incoming signal to and outgoing signal from VL_1 respectively.

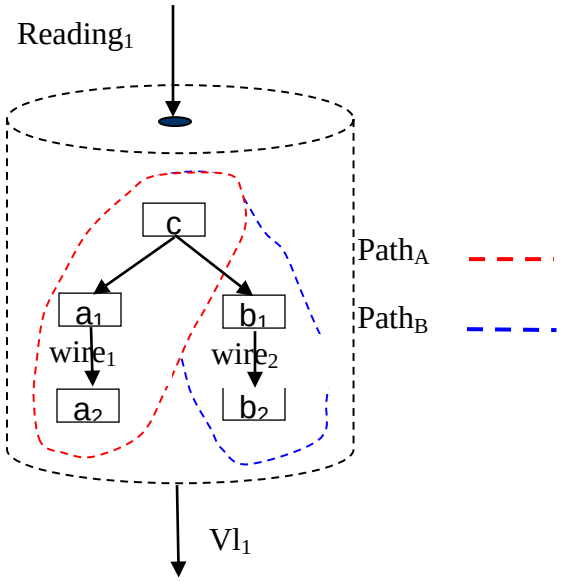

Figure 3.1: An example system.

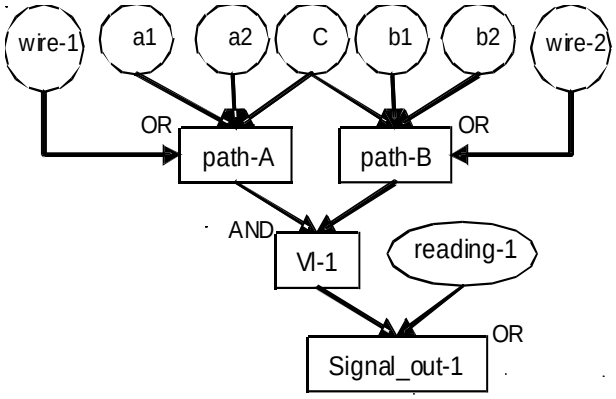

Figure 3.2: Fault- propagation graph corresponding to Figure 3.1.

For the system shown in Figure 3.1, our technique builds a graph as shown in Figure 3.2. In Figure 3.1, a_1, a_2, b_1, b_2, c, and $reading_1$ are the physical entities of the system. Each of them is represented as nodes in our graph, which are represented as circles or ovals in Figure 3.2. VL_1, $Path_A$ and $Path_B$ are represented by auxiliary nodes in our graph, which are represented as rectangles in Figure 3.2. $Signal\_out_1$ is the fault

status of $Reading_1$ that is being transmitted over VL_1. Thus, $Signal\_out_1$ will be correct if both $Reading_1$ and VL_1 are correct. In other words, the fault status of $Signal\_out_1$ will be TRUE if the fault status of either $Reading_1$ or VL_1 is TRUE. Hence, $Signal\_out_1$ is an auxiliary node in our graph with transformation function OR. The correctness of VL_1 depends on the correctness of either $Path_A$ or $Path_B$. VL_1 represents two redundant paths here: the fault status of VL_1 will be TRUE if fault status of both $Path_A$ and $Path_B$ is TRUE. Hence, VL_1 is an auxiliary node in our graph with transformation function AND. Correctness of both $Path_A$ and $Path_B$ are dependent on the correctness of all of their components. Hence, both $Path_A$ and $Path_B$ are auxiliary nodes in our graph with transformation function OR.

3.3 Fault-Propagation Analysis

We have developed two types of analyses: forward analysis and backward analysis. Given a set of components that could be faulty and set of critical nodes, forward analysis computes the (possibly empty) subset of critical nodes to which that a fault or faults can propagate. Given a set of critical nodes, backward analysis computes sets of components such that if all components in any one of those sets become faulty, fault can propagate to one or more of the critical components.

3.3.1 Forward Fault-Propagation Analysis

The input to the forward analysis is a set of components that could be faulty and a set of critical nodes. The goal of the forward analysis is to compute the (possibly empty) subset of critical nodes to which a fault or faults can propagate. This analysis is performed using Algorithm 3.1.

Input: *faultyNodes*: set of nodes representing physical entities in which faults have been introduced
criticalNodes: set of critical nodes that should not become faulty
Output: *faultyCritical*: set (possibly empty) that contains critical nodes that become faulty because of fault propagation
Declare: *workList*: a list of nodes

```
            visited:  a set of nodes

1 method forward(faultyNodes, criticalNodes)
2    set the fault status of each node in the set faultyNodes to TRUE
3    workList ← empty_list()
4    add each node in the faultyNodes set to workList
5    visited ← empty_set()
6    while workList is not empty do
7        n ← remove a node from workList
8        if n (element) visited then continue
9        add n to visited
11       if n has no transfer function then
12          add all successors of n to workList
13          continue
14       endif
15       if transfer function of n is OR then
16            st = logical OR of fault statuses of all predecessors of n
17       else if transfer function of n is AND then
18            st = logical AND of fault statuses of all predecessors of n
19       endif
20       set st as the fault status of n

21       if st is TRUE then
22            if n is in criticalNodes then
23            add n to faultyCritical
24          endif
25          add all successors of n to workList
26       endif
27  return faultyCritical
```

Algorithm 3.1

Algorithm 3.1 is a variation of a standard graph-reachability algorithm [12]. The inputs to the algorithm are two sets: *faultyNodes* and *CriticalNodes*. *faultyNodes* is the set of nodes in which faults are introduced. *CriticalNodes* is the set of nodes that must not become faulty. The algorithm outputs *faultyCritical*, which is a subset of *CriticalNodes*, and contains all critical nodes to which a fault or faults can propagate. The algorithm maintains two data-structures: *workList* and *visited*. *workList* is a list of nodes that must be processed, and *visited* is a set that stores all nodes that have been processed. The algorithm traverses the graph starting from each node in *faultyNodes*. In line 2, the fault status of each node in *faultyNodes* is set to TRUE. In lines 3-4, *workList* is initialized to contain each node in the *faultyNodes*. While the *workList* is not empty, the while loop starting at line 6 removes a node from *workList* (line 7) and

processes the node. The algorithm ensures that each node is processed only once (lines 8-9). Thus, it eventually terminates for any input graph.

When a node that has a transfer function is processed, its fault status is computed from the fault statuses of its predecessors based on the node's transformation function. In lines 15-16, the fault status of a node with an OR transformation function is computed. The fault status of such a node is the result of the logical OR operation of the fault statuses of all of its predecessors. Similarly, the fault status of a node with an AND transformation function is computed in lines 17-18. The fault status of such a node is the result of the logical AND operation of the fault statuses of all of its predecessors. When a node that does not have a transfer function is processed, the only operation that is performed is to add the node's successor nodes to *workList*.

If the fault status of a node evaluates to TRUE, all of its successor nodes are put into *workList* for processing (line 25) because the fault statuses of those successor nodes may now evaluate to TRUE. Also, when the fault status of a critical node evaluates to TRUE, the node is added to the result set *faultyCritical*.

Algorithm 3.1 processes each node only once. Thus, the while loop at line 6 iterates n times. In each iteration, the algorithm performs constant work. Thus, the complexity of Algorithm 3.1 is O(n), where n is the number of nodes in the graph.

We illustrate Algorithm 3.1 using the fault-propagation graph shown in Figure 3.2. Suppose a fault is introduced in components that correspond to nodes a_1 and a_2. Thus, *faultyNodes* contains these two nodes, and their fault statuses will be set to TRUE at the beginning of the algorithm. Suppose Signal_out$_1$ is the only critical node, and thus, *crtiticalNodes* contain only Signal_out$_1$. Suppose further that the algorithm first processes node a_1 and then node a_2. Because those nodes have no transfer function, the algorithm simply adds their successor $Path_A$ to *workList*. When the $Path_A$ is processed, because it has an OR transformation function, its fault status is evaluated by taking the

logical OR of the fault statuses of its predecessors: $wire_1$, a_1, a_2, and c. Because the fault statuses of a_1 and a_2 are TRUE, the fault of status of $Path_A$ is set to TRUE. Now the successor of $Path_A$, which is VL_1, is added to *workList*. When VL_1 is processed, its fault status is computed by taking the logical AND of the fault statuses of its predecessors: $Path_A$ and $Path_B$. Although the fault status of $Path_A$ is TRUE, the fault status of VL_1 evaluates to FALSE because the fault status of $Path_B$ (the other predecessor of $Path_A$) is FALSE. Because VL_1's fault status evaluates to FALSE, its successors are not added to *workList*. At this point, the algorithm terminates and returns *faultyCritical* because *workList* is empty. In this example, the algorithm returns the empty set *faultyCritical* as the result, which means that the fault from nodes a_1 and a_2 did not propagate to any critical node.

3.3.2 Backward Analysis

Given a set of critical nodes, the goal of the backward analysis is to compute the set S of pairs and triples of nodes such that (1) a pair (m, n) is in S if a fault can propagate to one of the critical nodes when faults are introduced in both nodes m and n, and (2) a triple (m, n, p) is in S if a fault can propagate to one of the critical nodes when faults are introduced in nodes m, n, and p. We designed two algorithms to perform backward analysis: a naive algorithm and an algorithm that uses Datalog.

3.3.2.1 Naive Backward Analysis

The naive backward-analysis algorithm is shown in Algorithm 3.2. The input to the algorithm is *CirticalNodes*, which is the set of nodes that must not become faulty. The output of the algorithm is the set S, which is described in the previous paragraph. In line 2, S is initialized to empty set. In lines 3-7, for each pair of nodes (A,B) in the graph, the algorithm calls the method forward, which is defined in Algorithm 3.1, to check whether any of the critical nodes can become faulty as a result of introducing faults in nodes A

and B. If one or more critical node can become faulty, then the algorithm adds the pair (A,B) to the result set S. The algorithm processes every node triple in similar manner in lines 9-13. In line 15, the algorithm returns S.

```
Input: criticalNodes: set of critical nodes that should not become faulty
Output: S: a set of pairs and triples of nodes such that (1) A pair (m, n) is in S if fault
          can propagate to one of the critical nodes when faults are introduced in both
          nodes m and n, and (2) a triples (m, n, p) is in S if fault can propagate to one
          of the critical nodes when faults are introduced in nodes m, n, and p.

1  method backward(criticalNodes)
2    S ←empty_set()
3    foreach pair of nodes (A,B) do
4      result ← forward({A, B}, criticalNodes) )
5      if result is not empty then
6          add (A,B) to S)
7      endif
8    done
9    foreach triples of nodes (A,B,C) do
10     result ← forward({A, B, C}, criticalNodes) )
11     if result is not empty then
12         add (A,B,C) to S
13     endif
14   done
15   return S
```

Algorithm 3.2. Naive Backward Propagation Algorithm

If n is the number of nodes in the graph, then there are (n * (n-1))/2 node pairs, and (n * (n-1) * (n-2))/6 node triples. Thus, Algorithm 3.2 calls method *forward* (shown in Algorithm 3.1) ((n+1) * n * (n-1))/6 times because *forward* is called once for each node pair and each node triple. Because the complexity of each call to *forward* is O(n), the overall complexity of Algorithm 3.2 is $O(n^4)$.

3.3.2.2 Datalog-Based Backward Analysis

Datalog [11, 13] is a query and rule language for deductive databases that syntactically is a subset of Prolog. A *deductive* database is a database that can deduce (i.e., conclude additional facts) based on rules and facts stored in the (deductive) database. A Datalog program consists of a set of clauses. A *clause* is a head literal

followed by an optional body. A *body* is a comma separated list of literals. A clause without a body is called a *fact*, and a clause with a body is called a *rule*. The punctuation ':-' separates the head of a rule from its body. Intuitively, a clause means that the head of a clause holds if each of the literals in the body holds. For example, consider two relations: parent(P, X), and sibling(X, Y). Suppose parent(P, X) holds for P and X if P is a parent of X, and sibling(X, Y) holds for X and Y if X is a sibling of Y. Then, the following clause means that if P is a parent of X and P is a parent of Y, then X and Y are siblings.

sibling(X, Y) :- parent(P, X), parent(P, Y)

To perform backward analysis using Datalog, we represent the fault-propagation graph, information about which nodes are critical nodes, and fault-propagation rules in a Datalog program. The Datalog program is then evaluated by a Datalog engine. The result of evaluating the Datalog program is the set S of desired pairs and triples (as defined above).

In FauPA, we use the bddbddb Datalog engine [10]. The feature of bdddbbd that distinguishes it from other Datalog engines is that bddbddb internally uses binary decision diagrams (BDDs) to evaluate a Datalog program. BDDs are a data structure that can efficiently represent large relations and provide efficient set operations. Thus, BDDs let bddbddb efficiently represent and operate on extremely large relations—relations that are too large to represent explicitly.

We create the Datalog program as follows. In the fault-propagation graph, if there is an edge from node X to node Y, then we generate the following fact:

```
edge(X, Y).
```

For each node X that represents a physical entity (e.g., LRU, switch, wire), we generate the following fact (recall that these are the nodes in which faults can be introduced):

```
leaf(X).
```

For each auxiliary node Y that has OR transfer function, we generate the following the following fact:

```
auxnode(Y, OR).
```

Similarly, for each auxiliary node Y that has an AND transfer function, we generate the following :

```
auxnode(Y, AND).
```

To encode fault-propagation rules, we use three relations: singleton, pair, and triple. The singleton relation `singleton(P,A)` means that if a fault is introduced in node A, then the fault can propagate to node P.

```
singleton(P,A) :- A = P, leaf(A).                              (1)
singleton(P,A) :- auxnode(P,OR),edge(Q,P),singleton(Q,A).      (2)
singleton(P,A) :- auxnode(P,AND), edge(Q,P),edge(R,P),
                  singleton(R,A),singleton(Q,A),R!=Q.          (3)
```

Rule 1 above is trivial: if A is the same node as P and the fault can be introduced in A, then the fault can propagate to P. Rule 2 and Rule 3 encode how a fault can propagate to auxiliary nodes with OR and AND transformation functions, respectively. Rule 2 means that, when a fault is introduced in node A, the fault can propagate to auxiliary node P, which has an OR transformation function, if fault can propagate to Q and Q is a predecessor of P. Rule 3 means that when a fault is introduced in node A, the fault can propagate to auxiliary node P that has an AND transformation function if the fault can propagate from A to both of P's predecessors Q and R. We assume that in the underlying fault propagation, every node has exactly two predecessors.

The relation `pair(P,A,B)` means that if a fault is introduced in two nodes, A and B, then the fault can propagate to node P.

```
pair(P,A,B) :- auxnode(P,OR),edge(Q,P),pair(Q,A,B).            (1)
pair(P,A,B) :- auxnode(P,AND),edge(Q,P),edge(R,P),
               singleton(Q,A),singleton(R,B),R!=Q,A!=B.        (2)
```

```
pair(P,A,B) :- auxnode(P,AND),edge(Q,P),edge(R,P),
               pair(Q,A,B),singleton(R,A),R!=Q,A!=B.          (3)
pair(P,A,B) :- auxnode(P,AND),edge(Q,P),edge(R,P),
               pair(Q,A,B),singleton(R,B),R!=Q,A!=B.          (4)
pair(P,A,B) :- auxnode(P,AND),edge(Q,P),edge(R,P),
               pair(Q,A,B),pair(R,A,B),R!=Q,A!=B.             (5)
pair(P,A,B) :- !singleton(P,A),!singleton(P,B),A!=B.          (6)
```

Rule 1 specifies how a fault propagates to an auxiliary node with an OR transformation function. Rules 2-5 specify how a fault propagates to an auxiliary node with an AND transformation function. Table 3.1 shows the interpretation of these rules. In the table, Q and R are the predecessors of node P. Rule 6 filters out the elements that are also in the `singleton` relation.

Table 3.1: Interpretations of rules used to define the relation `Pair(P,A,B)`

Rule	P is faulty because:
1	Q is faulty because of faults introduced at both A and B
2	Q is faulty because of fault introduced only at A, and R is faulty because of fault introduced only at B
3	Q is faulty because of faults introduced at both A and B, and R is faulty because of fault introduced only at A
4	Q is faulty because of faults introduced at both A and B, and R is faulty because of fault introduced only at B
5	Q is faulty because of faults introduced at both A and B, and Q is faulty because faults introduced at both A and B

The relation `triple(P,A,B,C)` is defined similar to `Pair(P,A,B)`.

3.3.2.3 Empirical Comparison of Datalog-Based and Naive Backward Analysis

We performed an empirical study to compare the efficiency of our naive backward-analysis algorithm and the Datalog-based algorithm. In one of our largest models, we identified seven nodes as critical. For each of those nodes, we (1) ran implementations of the two algorithms to compute the sets of all pairs and triples of nodes from which a fault can propagate to the specific critical node, and (2) measured the

time taken by each algorithm.

Figure 3.3 shows the collected data. The horizontal axis gives the names of the seven critical nodes. The vertical axis shows the times taken by each of the two algorithms for each critical node. For example, in case of when LRU_0509_OUT4 is set as the critical node, then the naive algorithm takes about 100 seconds, and the Datalog-based algorithm takes about 10 seconds. Figure 3.3 shows that the Datalog-based algorithm is at least an order of magnitude faster than the naive algorithm.

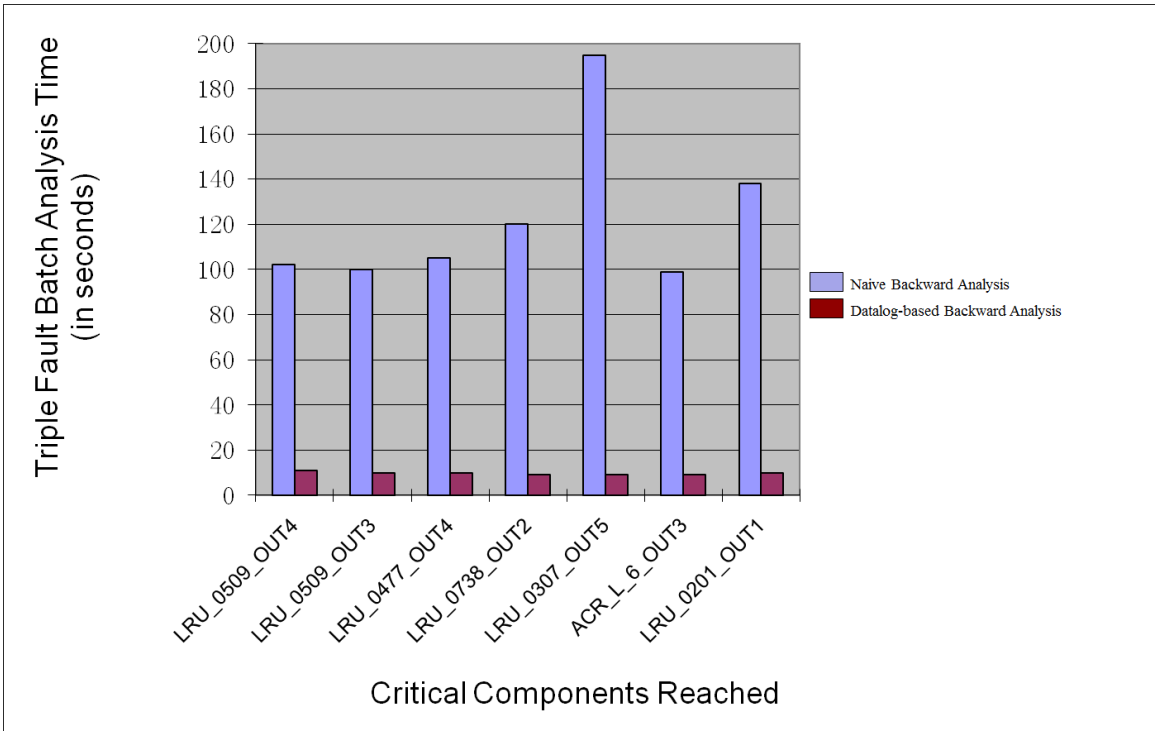

Figure 3.3: Time taken by Datalog-based and naive backward-analysis algorithms.

CHAPTER 4

COMMUNICATION SYSTEM MARKUP LANGUAGE (CSML)

This chapter describes Communication System Markup Language (CSML), an XML-based language to specify an AFDX-based communication system. The input to FauPA is CSML. FauPA automatically generates a fault-propagation graph that is used for fault-propagation analysis from the CSML specification of the model. The fault-propagation graph is used in the subsequent analysis stage.

4.1 CSML Specification

Table 4.1 shows the grammar for the CSML language in Backus-Naur form (BNF) [14]. A BNF specification is a set of derivation rules, written as

<symbol> **::=** __expression__

<symbol> is a *nonterminal*, and the __expression__ consists of one or more sequences of symbols; more sequences are separated by the vertical bar, '|', indicating a choice, the whole being a possible substitution for the symbol on the left. Symbols that never appear on a left side are *terminals*. On the other hand, symbols that appear on a left side are *non-terminals* and are always enclosed between the pair <>.

In a CSML specification, there are two types of root-level elements:

- `<of>` element
- `<vl>` element

4.1.1 <of> Element

For each hosted function of the system, there is an `<of>` element. An`<of>` element has a `name` attribute that stores the name of the function and helps in referring to a particular function. Each `<of>` elements has two types of elements as its children:

<host> element, <input> element, and <dest> element.

Table 4.1: Grammar of CSML in Backus-Naur Form (BNF).

```
<csml>  :=  "<csml>" <vl-list> <of-list> "</csml>"

<vl-list>  :=  <vl> | <vl> <vl-list>
<vl>  :=  "<vl" "name" "=" TEXT "source" "=" TEXT ">" <channel-list> <msg-list>
"</vl>"

<channel-list>  :=  <channel> | <channel> <channel-list>
<channel>  :=  "<channel>" <component-list> "</channel>"

<component-list>  :=  <component> | <component> <component-list>
<component>  :=  "<component>" TEXT "</component>"

<msg-list>  :=  <msg> | <msg> <msg-list>
<msg>  :=  "<msg" "name" "=" TEXT ">" <ds-list> "</msg>"

<ds-list>  :=  <ds> | <ds> <ds-list>
<ds>  :=  "<ds" "name" "=" TEXT ">" "<dest>" TEXT "</dest>" <param-list> "</ds>"

<param-list>  :=  <param> | <param> <param-list>
<param>  :=  "<param>" TEXT "</param>"

<of-list>  :=  <of> | <of> <of-list>
<of>  :=  "<of" "name" "=" TEXT ">" <host-list> <input-list> <dest-list> "</of>"

<host-list>  :=  <host> | <host> <host-list>
<host>  :=  "<host" "name" "=" TEXT ">" <output> "</host>"

<output>  :=  "<output>" TEXT "</output>"

<input-list>  :=  <input> | <input> <input-list>
<input>  :=  "<input>" <param-list> "</input>"

<dest-list>  :=  <dest> | <dest> <dest-list>
<dest>  :=  "<dest" "name" "=" TEXT "/>"
```

4.1.1.1 <host> Element

An <of> element has one or more <host> child elements. A <host> element has a name attribute, whose value identifies the computer or partition of the computer that computes this function. Each <host> element has an <output> child element.

An <output> element specifies the output signal computed by this <host> element for this <of> element.

4.1.1.2 <input> Element

An <of> element has one or more <input> child elements. The <input> elements are redundant with each other. An <input> element has one or more <param> elements as its children. The value of <param> elements inside each <input> element represents different readings of same environmental parameters. The number of <param> elements inside each <input> element is always same. A <param> child element specifies the input signal to this function <of> element.

4.1.1.3 <dest> Element

An <of> element must have one or more <dest> elements as children. A <dest> element represents the destinations or effectors who subscribes for the output signal computed by this <of> function. A <dest> element has a name attribute, whose value identifies the name of the effector who requires the computed output signal by this <of> function.

4.1.2 <vl> Element

For each virtual link of the system there is a <vl> element. Each <vl> element has two attributes: name and source. The value of the name attribute uniquely identifies the virtual link. The value of a source attribute uniquely identifies the component that sends messages through this virtual link. Each <vl> element has two types of elements as its children: <channel>, and <msg>.

4.1.2.1 <channel> Element

A <vl> element has one or more <channel> elements as children. Each of

these <channel> elements corresponds to a channel of the virtual link. Each channel element has one or more <component> elements as its children. A <component> element uniquely identifies a RDC, switch or wire of the system that is used in the virtual link.

4.1.2.2 <msg> Element

A <vl> element has one or more <msg> elements as children. Each of these <msg> elements corresponds to a message sent through this virtual link. Each <msg> element has one attribute name whose value uniquely represents the message. Each <msg> element has one or more <ds> elements as its children .

4.1.2.2.1 <ds> Element

A <ds> element represents one of the dataset associated with a message <msg> element. Each <ds> element must have one or more <param> element and <dest> element. A <param> element represents a parameter or signal packed in this dataset. A <ds> element must have one or more <dest> elements as children. A <dest> element represents the destination to which the signals packed inside this dataset are sent to.

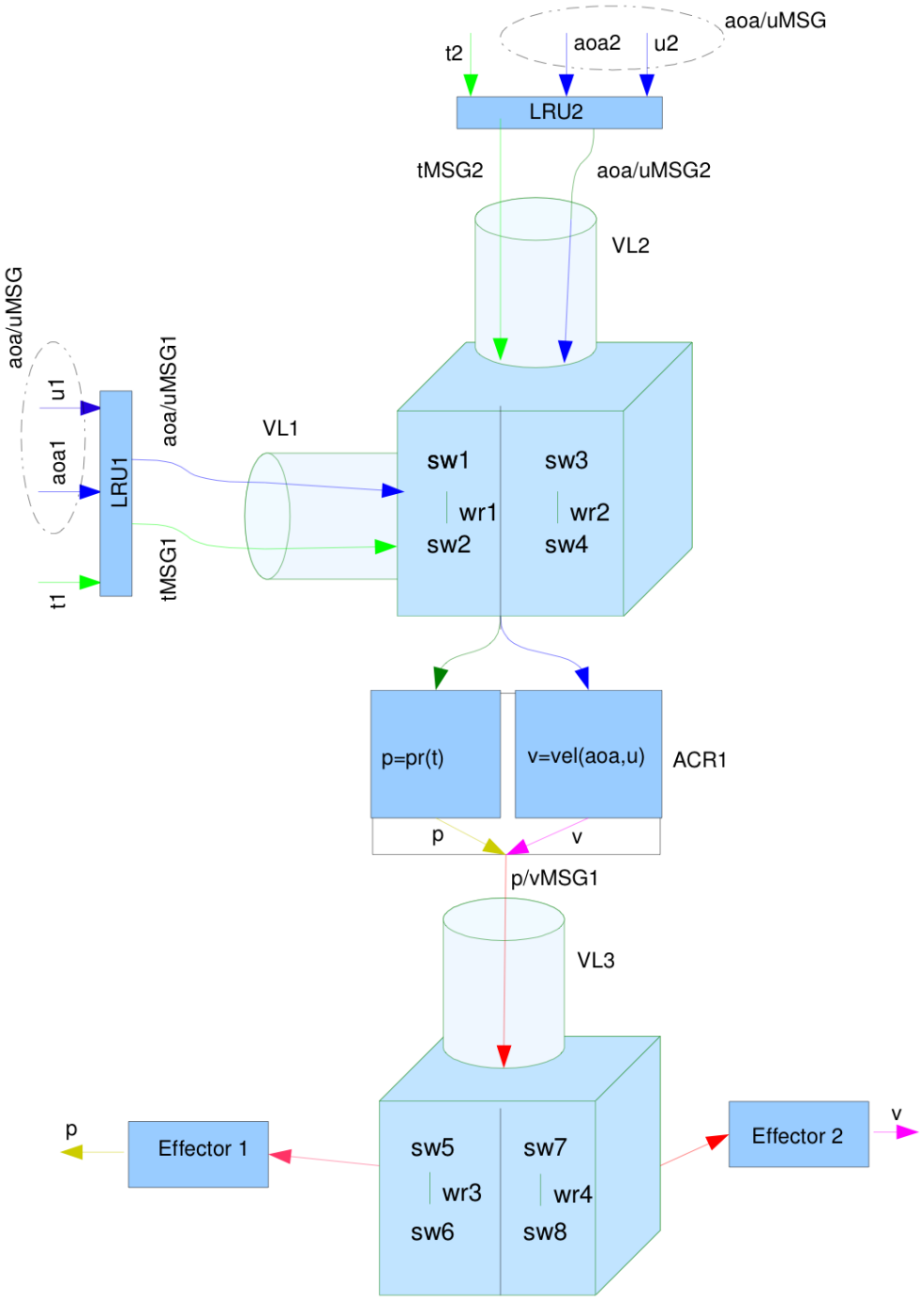

Figure 4.1: A sample AFDX-based system.

4.2 Sample Model with CSML Specification

In the following, we describe a sample system (shown in Figure 4.1) and then present its CSML specification (shown in Table 4.2). The descriptions of different parts of the system refer to the line numbers in the CSML specification where they are mentioned.

The sample system shown in Figure 4.1, there are two output functions, named `pr` and `vel`, hosted on the host or computer ACR1. The output function `vel` computes the velocity `(v)` of the plane. The inputs to `vel` are angle of attack (`aoa`) and speed(`u`). ACR1 receives two redundant copies of u: u1 and u2. ACR1 also receives two redundant copies of aoa: aoa1 and aoa2. The output function `pr` computes the pressure `(p)` inside the plane. The input to this function is temperature (`t`) inside the cabin. There are two redundant copies of temperature `t1` and `t2` available at ACR1. The output signal `p` is sent to the effector Effector1 and the output signal `v` is sent to the effector Effector2.

- Lines 2-13 of Table 4.2 define the output function `pr`. Because there are two redundant copies of the input parameter `t` available at ACR1, there are also two `<input>` elements.
- Lines 14-27 of Table 4.2 define the output function `vel`. Because there are two redundant copies of the input parameters `u and aoa` available at ACR1, there are also two `<input>` elements.

LRU1 and LRU2 are two sensors in this system that read the same data parameters: `t, aoa, and u`. The readings of LRU1 are `t1, aoa1, and u1`. The readings of LRU2 are `t2, aoa2, and u2`. These input readings are sent to ACR1 through two virtual links: `VL1` and `VL2`. Both virtual links consist of two channels. One of the channels of `VL1` contains switches sw1, sw2, and a wire wr1, and the other channel of `VL1` contains switches sw3, sw4, and a wire wr2. One of the

channels of VL2 contains switches sw5, sw6, and a wire wr3, and the other channel of VL1 contains switches sw7, sw8, and a wire wr4.

- Lines 28-39 of Table 4.2 specify the channels and source of the virtual link VL1.
- Lines 54-64 of Table 4.2 specify the channels and source of the virtual link VL2.

VL1 carries signals from the sensor LRU1 to ACR1. The input reading t1 is packed in a dataset called ds1, and sent through a message called tMSG1. Similarly, the input readings u1 and aoa1 are packed in a dataset called ds1, and sent through a message called u/aoaMSG1. Virtual link VL2 carries signals from sensor LRU2 to ACR1. The input reading t2 is packed in a dataset called ds1, and sent through a message called tMSG2. Similarly, the input readings u2 and aoa2 are packed in a dataset called ds1, and sent through a message called u/aoaMSG2.

- Lines 40-52 of Table 4.2 define the messages tMSG1 and u/aoaMSG1 along with the input readings packed inside them.
- Lines 65-77 of Table 4.2 define the messages tMSG2 and u/aoaMSG2 along with the input readings packed inside them.

ACR1 computes the output signals p and v from the input readings it received from VL1 and VL2. After computing the output signals, ACR1 sends both the output signals through the virtual link VL3. Virtual link VL3 consists of two channels. One of the channels of VL3 contains switches sw9, sw10, and a wire wr5, and the other channel of VL3 contains switches sw11, sw12, and a wire wr6. Lines 79-89 of Table 4.2 specify the channels and source of the virtual link VL3.

VL3 carries the output signals p and v from the source ACR1. These output signals are sent through the message p/vMSG1 to two effectors: Effector1 and

Effector2. The output signal `p` is packed in the dataset `ds1` and sent to Effector1. Similarly, the output signal `v` is packed in the dataset `ds2` and sent to Effector2. Lines 90-99 of Table 4.2 define the message `p/vMSG1` along with the output signals packed inside them.

Table 4.2: CSML specification of the sample system.

```
<csml>
  <of name="pr">
    <host name="ACR1">
      <output>p</output>
    </host>
    <input>
      <param>t1</param>
    </input>
    <input>
      <param>t2</param>
    </input>
    <dest name="Effector1"/>
  </of>
  <of name="vel">
    <host name="ACR1">
      <output>v</output>
    </host>
    <input>
      <param>aoa1</param>
      <param>u1</param>
    </input>
    <input>
      <param>aoa2</param>
      <param>u2</param>
    </input>
    <dest name="Effector2"/>
  </of>
  <vl name="VL1" source="LRU1">
    <channel>
      <component>sw1</component>
      <component>wr1</component>
      <component>sw2</component>
    </channel>
    <channel>
      <component>sw3</component>
      <component>wr2</component>
      <component>sw4</component>
    </channel>
    <msg name="tMSG1">
      <ds name="ds1">
        <dest>ACR1</dest>
        <param>t1</param>
      </ds>
    </msg>
    <msg name="aoa/uMSG1">
      <ds name="ds1">
        <dest>ACR1</dest>
        <param>aoa1</param>
        <param>u1</param>
      </ds>
    </msg>
  </vl>
  <vl name="VL2" source="LRU2">
    <channel>
      <component>sw1</component>
      <component>wr1</component>
      <component>sw2</component>
    </channel>
    <channel>
      <component>sw3</component>
      <component>wr2</component>
      <component>sw4</component>
    </channel>
    <msg name="tMSG2">
      <ds name="ds1">
        <dest>ACR1</dest>
        <param>t2</param>
      </ds>
    </msg>
    <msg name="aoa/uMSG2">
      <ds name="ds1">
        <dest>ACR1</dest>
        <param>aoa2</param>
        <param>u2</param>
      </ds>
    </msg>
  </vl>
  <vl name="VL3" source="ACR1">
    <channel>
      <component>sw5</component>
      <component>wr3</component>
      <component>sw6</component>
    </channel>
    <channel>
      <component>sw7</component>
      <component>wr4</component>
      <component>sw8</component>
    </channel>
    <msg name="p/vMSG1">
      <ds name="ds1">
        <dest>Effector1</dest>
        <param>p</param>
      </ds>
      <ds name="ds2">
        <dest>Effector2</dest>
        <param>v</param>
      </ds>
    </msg>
  </vl>
```

CHAPTER 5

GUI BASED DISPLAY

The GUI-based display of FauPA software lets the user perform a number of fault-propagation tasks, including (1) determining whether faults in manually-selected components can propagate to the critical airplane components, (2) determining whether faults in automatically-selected components (currently single components, pairs of components and triples of component are implemented) can propagate to the critical airplane components, and (3) viewing both textually and graphically the paths in the airplane through which faults can propagate to the critical airplane components.

Figure 5.1 shows the main screen of FauPA, which has three parts. First, the view in the upper left part of the screen shows one type of hierarchical view of the system under analysis. In this hierarchical view, physical components are shown as the terminal nodes, and a non-terminal node represents a subsystem. Second, the view in the bottom left part of the screen shows another type of hierarchical view of the system. In this hierarchical view, each terminal node represents a dataset, which consists of multiple data signals (not shown). A non-terminal node represents either a virtual link or a message. Nodes representing messages are shown as children of a node that represents the virtual link through which those messages are transmitted. Nodes representing datasets are shown as children of a node that represents the message that contains those datasets. Third, in the right part of the screen, FauPA displays the results of the fault-propagation analyses.

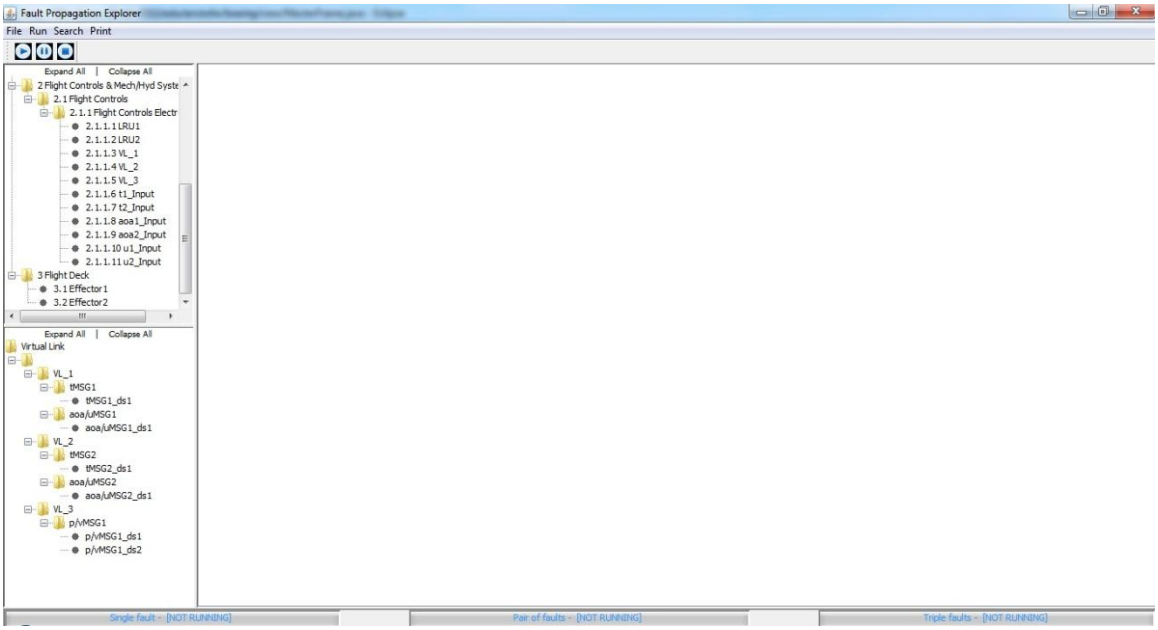

Figure 5.1: Main view of FauPA.

5.1 Performing Fault Propagation Tasks

FauPA provides two ways to perform the fault-propagation tasks: (1) manual and (2) batch. In batch mode, a fault can be set automatically in every single component, pair of components, and triples of components at a time. FauPA also lets all these kinds of propagation tasks to be run simultaneously. Thus, for example, a user can start a manual test while a batch test is running.

5.1.1 Manual Propagation Task

For the manual propagation task, a user introduces faults on some set of physical components in the system, initiates the propagation, and FauPA determines whether any of those faults propagates to the critical components. FauPA displays the results after the propagation is completed.

5.1.2 Batch Propagation Task

In the batch single-fault propagation task, FauPA automatically introduces faults in each component, one at a time, and determines whether the fault propagates to any critical component(s). In batch pair-of-faults mode, FauPA automatically introduces faults in every pair of components and determines whether the faults reach any critical components. Similarly, in batch triple-of-faults mode, FauPA automatically introduces faults in every triple of components and determines whether the faults reach any critical components. The list of all single, pair, and triple components from which faults can propagate to a critical node is displayed in the right side of the FauPA screen as shown in Figure 5.2.

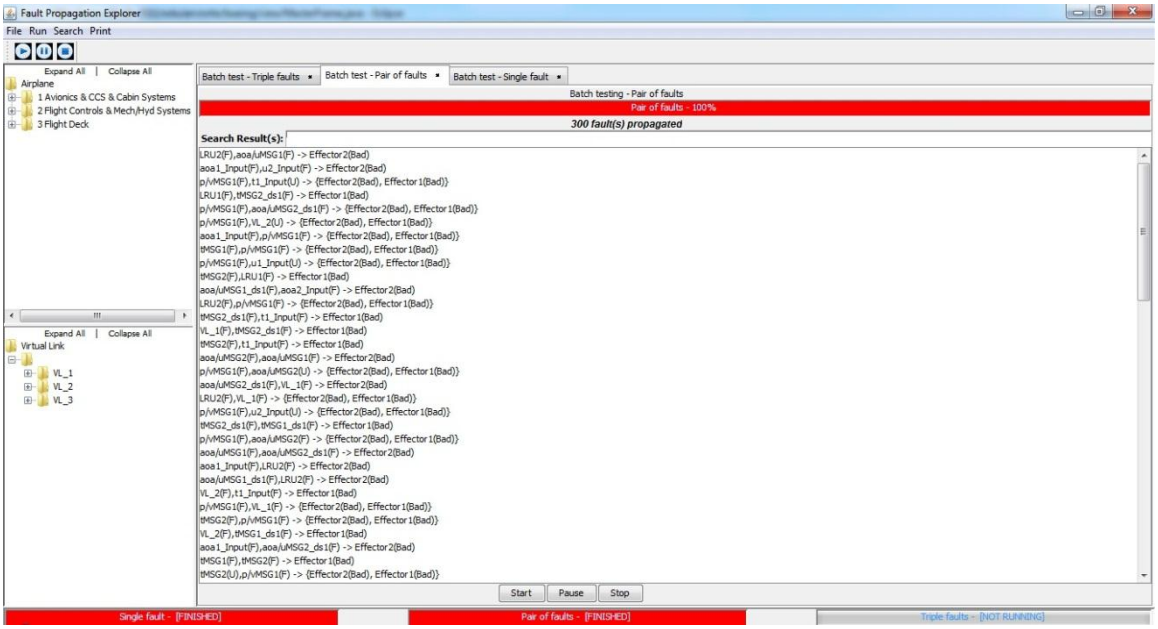

Figure 5.2: Results of batch fault-propagation task.

5.2 Visualizing the Fault Propagation Results

If the fault(s) propagates to any critical component, the propagation information is displayed on the right side of the screen. The results are displayed in two views: textual

and graphical.

5.2.1 Textual View of Analysis Results

The textual view shows the information about the components through which the fault has propagated to reach the critical component(s). Figure 5.3 shows a fault propagation result that is being displayed in textual view. Figure 5.3 shows for a particular case, fault propagated through two LRUs, one ACR, eight switches, and three VLs before reaching a critical node.

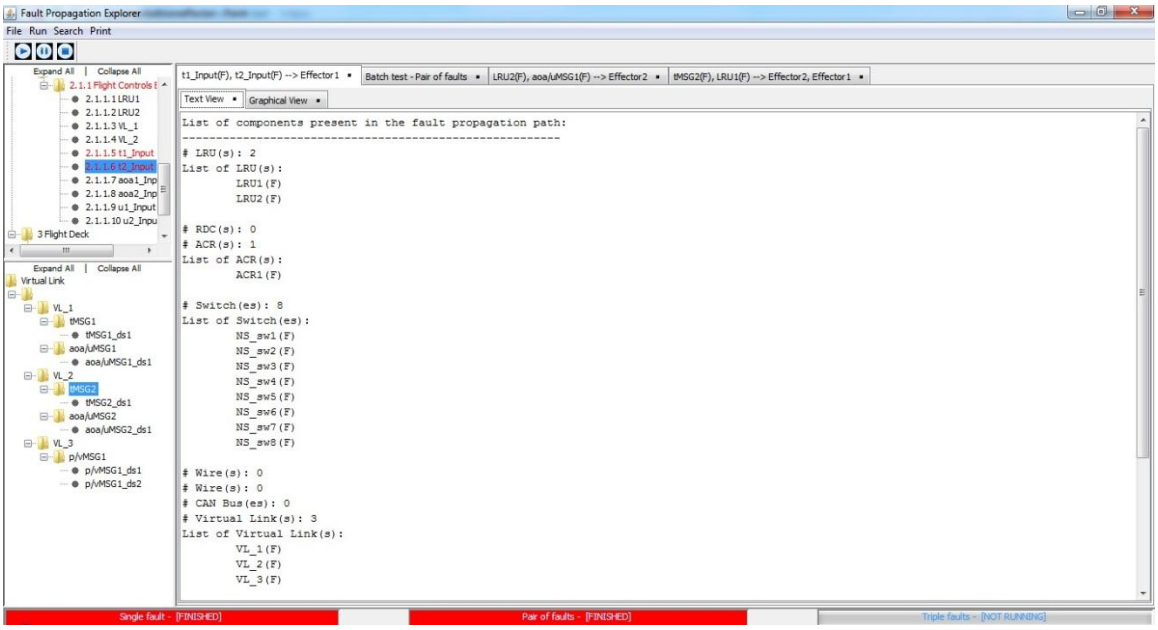

Figure 5.3: Textual view of the analysis results.

5.2.2 Graphical View of Analysis Results

The graphical view shows the information about the components through which the fault has propagated as a graph that represents the model at two levels: subsystem-level view and detailed view.

5.2.2.1 Subsystem-level graphical view of analysis results

Figure 5.4 shows the subsystem-level graphical view of the analysis results. In this view, every node in the graph represents a subsystem. Nodes that are involved in fault propagation are colored differently: (1) nodes containing the components where fault is introduced and nodes containing the affected critical airplane component are shown in **RED** and (2) nodes involved in the propagation are shown in **ORANGE**. Figure 5.4 shows that a fault that is introduced in Flight Control System propagated through the components in Avionics & CCS & Cabin Systems, and finally reached a critical airplane-level component in Flight Deck Systems. The nodes that contain the affected critical airplane components are shown in circle whereas other nodes are shown in rectangle shape.

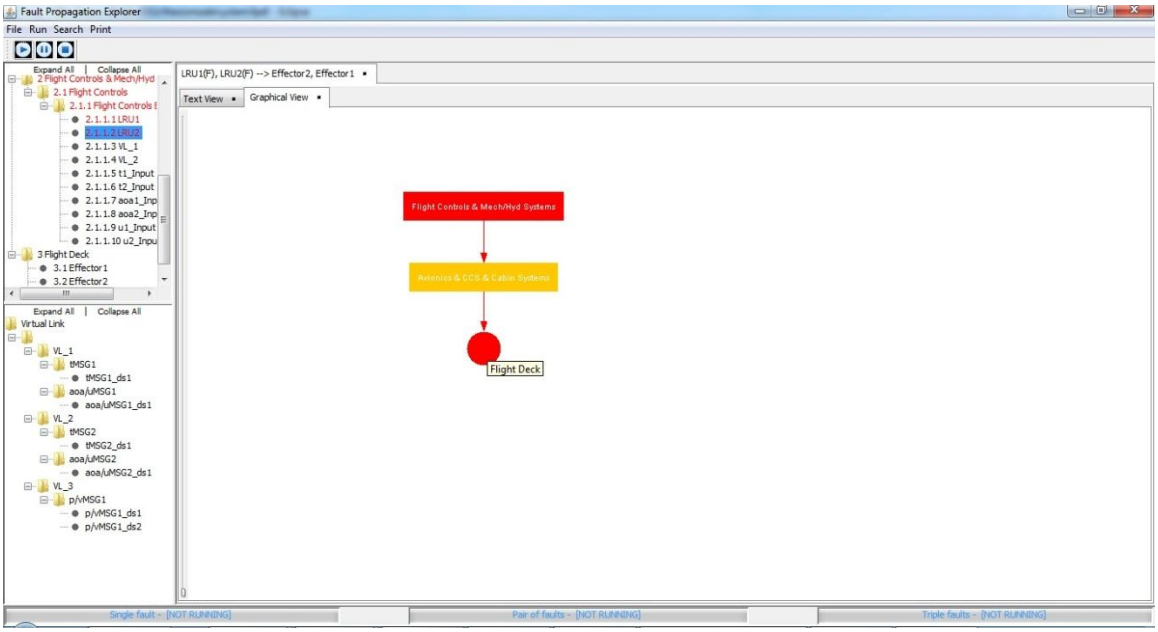

Figure 5.4: Subsystem-level graphical view of analysis results.

5.2.2.2 Detailed Graphical View of Analysis Results

Figure 5.5 shows the detailed graphical view of the analysis results. This view displays all physical components, VLs and input signals as nodes. The coloring and

shapes of the nodes are similar to the subsystem-level graphical view described in section 5.2.2.1. Additionally, because this view represents all components of the system under consideration, those nodes that are *not* involved in the propagation are shown in **GRAY**. In Figure 5.5 one of the critical airplane components is shown in grey because fault did not propagate there.

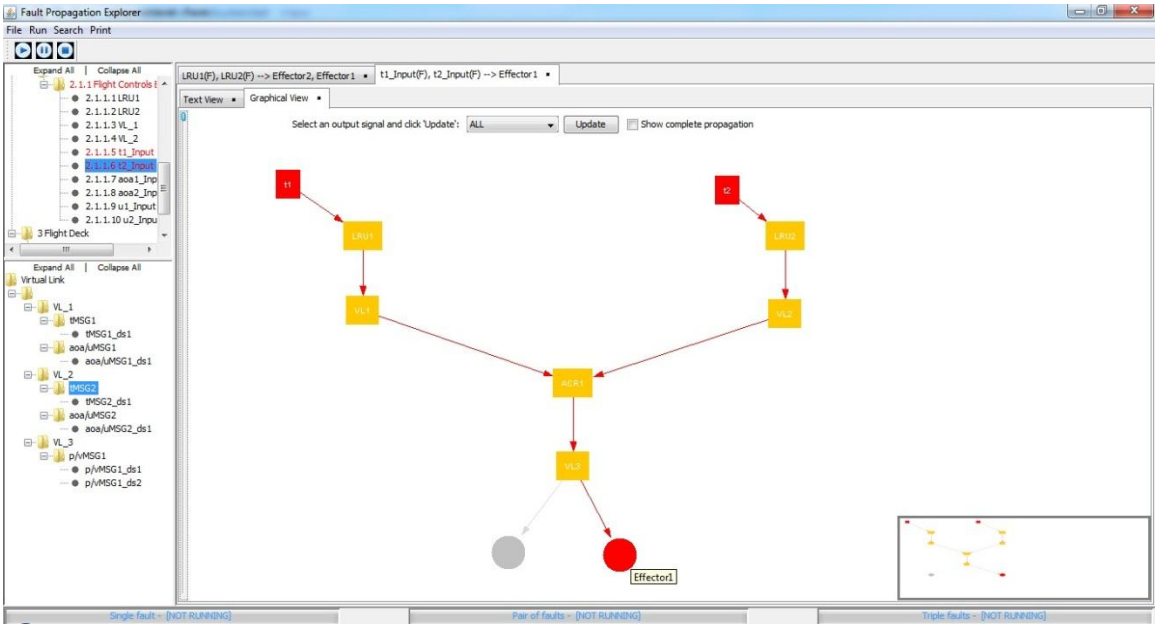

Figure 5.5: Detailed graphical view of analysis results.

The channels of a virtual link are not displayed by default in graphical views. However, clicking on a virtual link shows its channels as shown in Figure 5.6. The graphical views also support two other features that make it easier to navigate across views of large systems. The first feature is the mini-map feature. The mini-map, which is shown at the bottom right corner of a graphical view, shows a miniature view of the entire view. When a user clicks at a certain part of the system in the mini-map, the main view is centered on that part of the system. The second feature is the zoom-in and zoom-out feature that lets a user see the graph at different level of granularity.

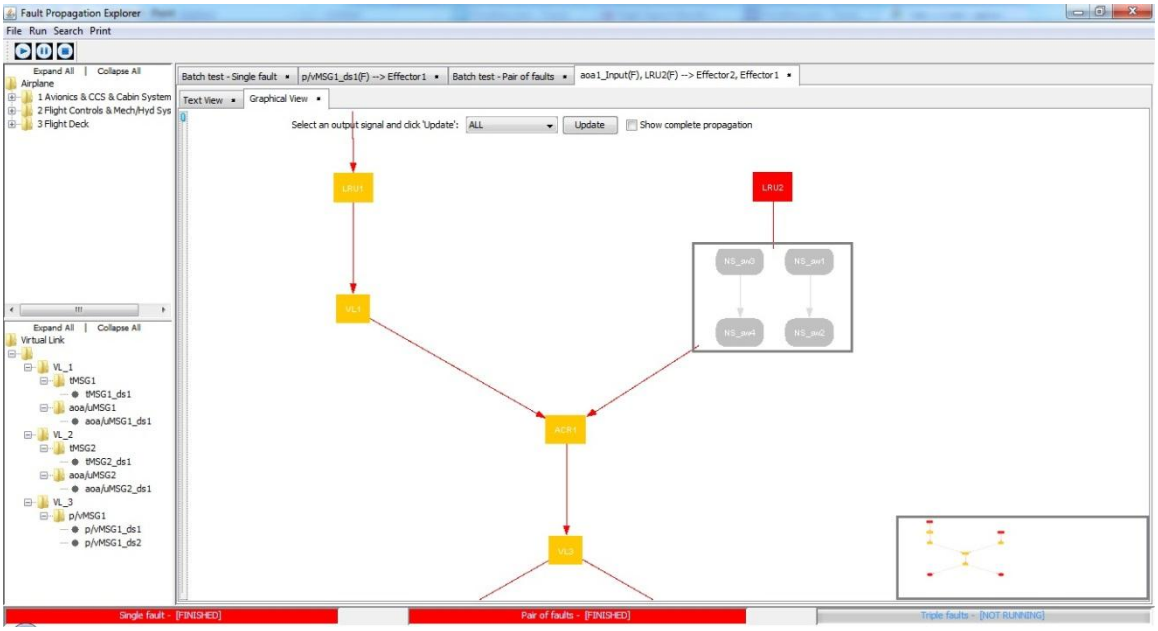

Figure 5.6: View showing the channels inside a virtual link.

CHAPTER 6

CONCLUSION

In safety-critical, networked embedded systems, it is important that the way in which a fault in one component can propagate through the system is analyzed correctly. Many real-world systems such as modern aircraft and automobiles use large-scale networked, embedded systems with complex behavior. Because of the large size of such systems, it t is challenging to manually analyze the way in which a fault or faults propagates through such systems. Furthermore, these systems usually have characteristics such as sharing of network or computational units among different sub-systems that make it even more challenging to manually analyze those systems.

In this work, we have developed techniques and a software tool, FauPA, that uses those techniques to automate fault-propagation analysis of large-scale, networked embedded systems such as those used in modern aircrafts. This work makes three contributions. First, we developed algorithms for two types of analyses: a forward analysis and a backward analysis. For backward analysis, we developed two techniques: a naive algorithm and another algorithm that uses Datalog. Second, we developed a language that we call Communication System Markup Language (CSML) based on XML. A system can be specified concisely and at a high-level in CSML. Third, we developed a GUI to visualize the system that is specified in CSML. The GUI also lets the user visualize the results of fault-propagation analyses.

FauPA has two important advantages over the state-of-the-art fault-propagation approach that uses fault-trees. First, the input to FauPA is not a fault-tree, which can be difficult to construct manually for large systems. In particular, networked embedded systems that share network and computational units across sub-systems have a graph topology. Thus, in a fault-tree based approach, the user must convert the graph topology

to fault-trees. In contrast, the input to FauPA is a high-level specification of the system in CSML, and FauPA automatically constructs the fault-propagation graph (similar to fault-trees) from that specification. Second, FauPA's fault-propagation analyses can be more efficient than those used by fault-tree-based tools because the underlying fault-propagation model used by FauPA's analyses is a graph, which matches the natural representation (i.e., graph topology) of the system.

There are several ways in which this work can be extended. First, current fault-propagation analyses do not support timing-related constraints that exist in networked, embedded systems. One extension to this work is to address this shortcoming of current analyses. Second, the current analyses support only OR and AND transfer functions. In the future, the analyses can be extended to support other types transfer functions that can aggregate fault statuses differently. Third, building a CSML specification from scratch can be error-prone and difficult. In future, a GUI-based tool can be developed that can assist a user to develop a CSML specification through a series of dialog boxes. Finally, this work focuses on networked, embedded systems that are used in modern aircrafts and that use AFDX communication network. In the future, FauPA can be extended to other domains such as automobiles that use FlexRay communication network.

REFERENCES

[1] “AFDX”, http://en.wikipedia.org/wiki/Avionics_Full-Duplex_Switched_Ethernet (Accessed October 22, 2011)

[2] “FTA”, http://en.wikipedia.org/wiki/Fault_tree_analysis (Accessed October 22, 2011)

[3] “FlexRay”, http://en.wikipedia.org/wiki/FlexRay (Accessed October 22, 2011)

[4] “FMEA”, http://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis (Accessed October 22, 2011)

[5] “ARINC”, http://en.wikipedia.org/wiki/ARINC (Accessed October 22, 2011)

[6] “TTP”, http://en.wikipedia.org/wiki/Time-Triggered_Protocol (Accessed October 22, 2011)

[7] "NASA Evaluate AFDX and ARINC 653 for Exploration Vehicle," SYSGO Press Release.

[8] SCHAADT, D., "AFDX/ARINC 664 Concept, Design, Implementation, and Beyond," SYSGO AG.

[9] “IMA”, http://en.wikipedia.org/wiki/Integrated_modular_avionics (Accessed October 22, 2011)

[10] “BDDBDDB”, http://suif.stanford.edu/bddbddb (Accessed October 22, 2011)

[11] “Datalog”, http://en.wikipedia.org/wiki/Datalog (Accessed October 22, 2011)

[12] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, Charles L. Leiserson.

Introduction to Algorithms. McGraw-Hill Higher Education. 2001.

[13] S. Das. Deductive Databases and Logic Programming. Addition-Wesley, 1992.

[14] J . W. Backus. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. Proceedings of the International Conference on Information Processing, UNESCO, 1959, pp.125-132.